

# JavaScript

About Me:

**Ralph Zajac**

[ralph@TheOrangelT.com](mailto:ralph@TheOrangelT.com)

<http://TheOrangelT.com/presentations>



# So what's so interesting in JavaScript?

- Objects inherit directly from other objects.
- Prototypal inheritance.
- Nested functions.
- Functions as data.
- Loosely typed language



# Build in types.

- **Number**
- **String (immutable)**
- **Boolean**
- **Function**
- **Object**
- **Array**
- **null**
- **undefined**
- **Date**
- **RegExp**
- **Error**



# Numbers

All numbers are 64-bit floating-point values.

Hexadecimal values start with '0x' or '0X'.

Octal values start with 0 (zero) but not all implementations of JavaScript support it so you should not use numbers starting with 0.



# Numbers. Predefined constants.

`Number.MAX_VALUE` – Largest representable number  
`Number.MIN_VALUE` – Smallest (closest to zero) number  
`Number.NaN` – **Special not a number.** `isNaN('string');`  
`NaN`  
`Infinity`  
`Number.POSITIVE_INFINITY`  
`Number.NEGATIVE_INFINITY`



# Numbers. Useful methods.

```
var iNumb = 123456.789;  
iNumb.toString(radix);  
iNumb.toFixed(n);  
    iNumb.toFixed(0); // 123456  
    iNumb.toFixed(2); // 123456.79  
iNumb.toExponential(n);  
    iNumb.toExponential(1); // 1.2e+5  
    iNumb.toExponential(3); // 1.235e+5  
iNumb.toPrecision(n);  
    iNumb.toPrecision(4); // 1.235e+5  
    iNumb.toPrecision(7); // 123456.8
```



# Converting Strings to Numbers

```
var iNumber = parseInt('123', 10);  
var iNumber2 = parseFloat('123.456');  
  
parseInt('12.34', 10); // 12  
parseInt('something', 10); // NaN  
parseInt('12.34 something', 10); // 12  
parseFloat('12.34 something'); // 12
```

Both functions return any number at the beginning of a string, ignoring any trailing non-numbers.

```
if (!isNaN(iNumber)) {  
    ...  
}
```



# undefined, undeclared and null

```
var a,           // undefined (declared but unassigned)
    b = undefined, // same as a
    c = null;
// var d;       // undeclared
```

In boolean context undefined and null evals to false.

```
if (a) {...} //false
if (b) {...} //false
if (c) {...} //false
```

```
if (d) {...} // error (script stops)
```

```
if(a == b) { ... } // true
if(a === b) { ... } // true
```

```
if(a == c) { ... } // true
if(a === c) { ... } // false
```



# Strings

```
var sAla = "Ala"; // Double quotes or  
var sMa = ' ma'; // single quotes allowed  
var sKot = ' kota.';  
  
var sSentence = sAla + sMa + sKot;  
  
var iLength = sSentence.length;  
  
var sLastDot =  
    sSentence.charAt(sSentence.length - 1);  
  
var sSub = sSentence.substring(1, 4);  
var iFind = sSentence.indexOf('ma');
```



# Arithmetic operators

- Addition (+)
- Subtraction (-)
- Multiplication (\*)
- Division (/)
- Modulo (%)
- Unary minus (-)
- Unary plus (+)
- Increment (++)
- Decrement (--)



# Equality Operators

- Equality (==) and identity (===)
- Inequality (!=) and (!==)



# Relational Operators

- Less then (<)
- Greater then (>)
- Less then or equal (<=)
- Greater then or equal (>=)



# Logical operators

- Logical AND (&&)
- Logical OR (||)
- Logical NOT (!)



# Bitwise Operators

- Bitwise AND (&)
- Bitwise OR (|)
- Bitwise XOR (^)
- Bitwise NOT (~)
- Shift left (<<<)
- Shift right with sign (>>>)
- Shift right with zero fill (>>>>)



# Assignment Operators

- Assignment (=)

## Assignment with operation

=, +=, -=, \*=, /=, %=, <<=, >>=,  
>>>=, &=, |=, ^=



# Miscellaneous Operators

- Ternary operator (`? :`)
- The object creation operator (`new`)
- The `delete` operator
- The `void` operator
- The comma operator (`,`)
- The array access operator (`[]`)
- The object access operator (`.`)
- The function call operator (`()`)



# The `in` Operator

It evaluates to `true` if the left value is the name of a property of the right side object.

Inherited properties evaluate to `true` unlike in case of `hasOwnProperty()` method.

```
var oPoint = {iX: 10, iY: 20};  
'iX' in oPoint; // true  
'toString' in oPoint; // true
```



# The instanceof operator

The operator evaluates to `true` if the left-side object is an instance of the right-side class.

```
var oD = new Date();
```

```
oD instanceof Date // true
```

```
oD instanceof Object // true
```

```
oD instanceof Number // false
```

```
var aA = [1, 2, 3, 4];
```

```
aA instanceof Array // true
```

```
aA instanceof Object // true
```

```
aA instanceof RegExp // false
```



# The typeof operator

Evaluates to:

'string' for string values.

'number' for number values.

'boolean' for boolean values.

'object' for objects, arrays and `null` (!)

'function' for function operands

'undefined' if the operand is undefined.

```
if(typeof value == 'string'){
    return "'" + value + "'";
} else {
    return value;
}
```



# Statements

- break
- case
- continue
- default
- do/while
- **The Empty statement (;)**
- for
- for/in
- function
- if/else
- **Labels**
- return
- switch
- throw
- try/catch/**finally**
- var
- while
- **with**



# Primitive and reference types

In JavaScript we have two primitive types:

- Numbers
- Booleans

Reference types:

everything else - **except String**



# Objects

All objects in JavaScript inherit from the Object class.

```
var oEmpty = {};  
var oEmpty = new Object();  
var oPoint = {iX: 10, iY: 20};  
var oPoint2 = {  
  iX: 10,  
  iY: 20,  
  distance: function(oPoint){  
    console.log(this.iX); // 10  
  }  
};
```

```
var oOriginal = {};  
oOriginal.iNewProperty = 5;  
oOriginal.newFunc = function (a, b){...}
```



# Checking object property existence

```
var o = {},  
    b = {foo: 'ala', bar: undefined},  
    c = {fooc: null};  
  
if('foo' in b) { ... } // true  
if('bar' in b) { ... } // true  
if('big_bar' in b) { ... } // false  
  
if(o.something === undefined){ ... } // true  
  
if (o.something) {  
    // property exists  
} else {  
    // property does not exist  
}
```



# Checking object vs. variable existence

```
var oObj = {};  
if (oObj.somePropertyOrMethod) { .. } // false
```

As long as oObj is declared as an object.

```
// var oObj2;  
if (oObj2.somePropertyOrMethod) { .. } // error
```

Remember you can't check existence of a variable in the same way

```
//var d;  
if (d) {...} // error (script stops)  
  
if(typeof aaaa !== 'undefined'){ ... }
```



# Objects may act like associative arrays.

```
var oObj = {  
    sPrperty1: 'a',  
    iProperty2: 20,  
    method: function(){}  
}  
  
oObj.sProperty1 === oObj['sProperty1'];  
oObj.method === oObj['method'];  
  
if(oObj.sProperty1 === oObj['sProperty1'])  
    {...}; // true  
  
for (sProp in oObj) {  
    console.log(sProp);  
} // sPrperty1, iProperty2, method
```



# PHP and JS data exchange

## PHP

```
json_encode( mixed $value );  
json_decode( string $json [, bool $assoc] );
```

## JavaScript

<http://www.json.org/json2.js>

```
var sStr = o.toJSON( object );  
var oObj = JSON.parse( string );
```



# Universal Object Properties and Methods

- constructor

```
var oD = new Date();  
oD.constructor === Date; // true  
if((typeof oD === 'object') && (oD.constructor ===  
    Date)) { ... }  
if((typeof oD === 'object') && (oD instanceof Date))  
    { ... }
```

- toString()
- valueOf()
- hasOwnProperty()
- propertyIsEnumerable()
- isPrototypeOf()



# `in` operator vs. `hasOwnProperty()` method

With `in` operator inherited properties evaluate to `true` with `hasOwnProperty()` only ones that are defined in the object itself.



# Arrays

```
var aMyArray = [];  
var aMyArray = new Array();  
  
var aMyNumb = [1,2,3,'ala', 2.5];  
var aNewArray = new Array(1,2,3,4, 'test');  
  
console.log(aMyNumb[3]); // ala  
aMyArray[5] = 5;  
  
aMyNumb[0] = [];  
aMyNumb[0][0] = 5;
```



# Array methods and properties

- `join('separator');`
- `reverse();`
- `sort(function);`
- `concat(Array);`
- `slice(n[, m]);`
- `splice(n[, m]);`
- `push(); pop();`
- `unshift(); shift();`
- `toString();`
- `Array.length`



# Functions. What's different?

## Nested functions:

```
function calculate(a,b) {  
    function square(x) { return x*x };  
    return square(a)+b;  
}
```

## Function literals (unnamed functions):

```
var f = function(x) { return x*x };  
f(5); // 25  
var g = f;  
g(5); // 25
```

**Lexical scoping.**

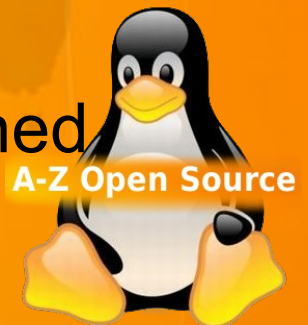


# Function arguments object

Within a body of a function you have an access to `arguments` identifier. It's **array like** object known as **Arguments object** that allows you to retrieve function arguments by number rather than name.

```
function ala(a, b, c, d, e){
    console.log(arguments.length);
    console.log(arguments[2]);
}
ala(1, 2, 3); // 3
ala(1, 2, 3, 4); // 4
ala(1, 2, 3, 4, 5, 6); // 6
```

When you skip a function's argument it's assigned **undefined value**.



# Function arguments object (cont.)

Arguments object also defines `callee` property that refers to the function that is currently executed.

Why do we need it?  
unnamed functions

```
var fnSum = function (x) {  
    if( x <= 1) return 1;  
    return x + arguments.callee(x-1);  
}  
fnSum(5);  
fnSomething = fnSum;  
fnSomething(5);
```



# Number of expected arguments

```
function ala(a, b, c, d, e) {  
    if(arguments.callee.length == 5) {  
        ...  
    } else {  
        ...  
    }  
}
```



# SCOPE



A-Z Open Source

# Lexical scope

In JavaScript functions are lexically scoped rather than dynamically scoped. That means that they run in the scope in which they were defined, not from which they are executed.

When the JavaScript interpreter invokes a function, it first sets the scope to the scope chain that was in effect when the function was defined.

Next it adds a new object known as *call object* to the front of the scope chain.

The call object is initialized with a property named `arguments` that refers to Arguments object for the function.

Named parameters of the function are added to the call object next.

Any local variables declared with the `var` statement are also defined within this object.



# Global Object

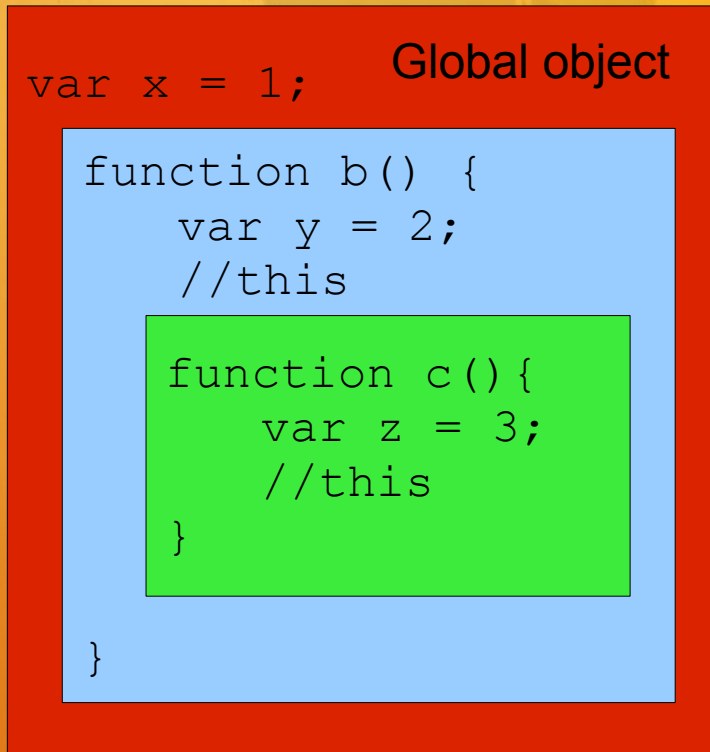
Top of the food chain.

Father of all global methods and properties.

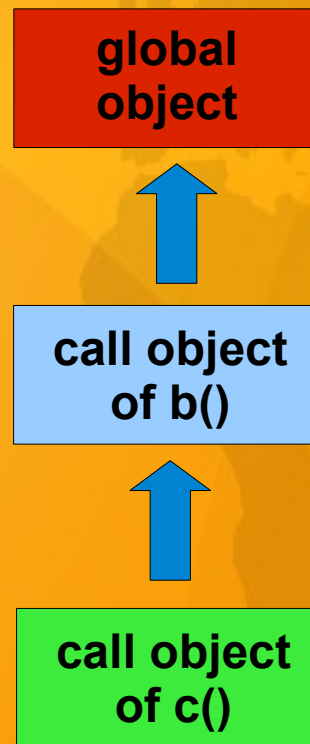


# Scope

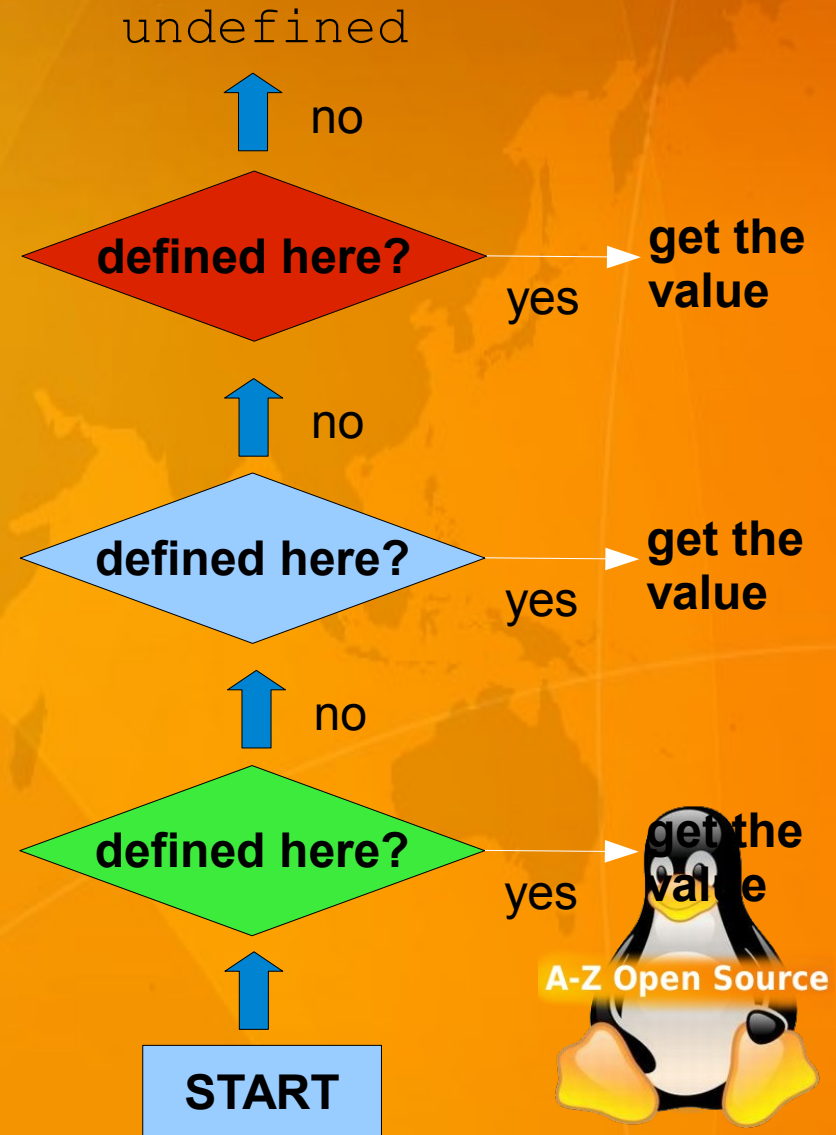
## Lexical scope



## Scope chain



## Variable lookup



# Variable scope

## Global scope:

Everything declared outside of functions,

No need to declare variables with `var` statement

## Local scope:

Variables declared inside a function + its parameters

Local variable takes precedence over global variable with the same name.

Variables declared without `var` statement in a body of a function get global scope.



# No Block Scope

Variables declared in body of a function are defined throughout the function.

```
var sScope = 'global';  
function f() {  
    console.log(sScope); // undefined  
    var sScope = 'local';  
    console.log(sScope); // 'local'  
}  
f();
```



# Closures

```
function ala(iNumber) {  
    var iInside = iNumber+5; } Private parameter  
  
    return function() {  
        console.log(iInside + ' ' + iNumber); } Nested function  
    }  
}
```

```
var fnFoo = ala(5);  
foo(); // 10 5
```



# Constructor function

```
function Point(iX, iY) { // Constructor function
    this.iX = iX;
    this.iY = iY;
}
```

```
var oMyLocation = new Point(10, 20);
```

Constructor function provides name for a “class” of objects and initializes properties, such `iX` and `iY`, that may be different for each instance of the class.



# The `new` operator

The `new` operator must be followed by a function (called *Constructor Function*). It creates a new empty object, and then invokes the function, passing the new object as the value of the `this` keyword.

The constructor function job is to initialize a newly created object's properties before the object can be used.

Methods of an object should be defined using `prototype` property of the object.



# The `prototype` property

Every JavaScript object includes an internal reference to another object, known as its *prototype object*. Any properties or methods of the prototype object appear to be properties of an object for which it is the prototype.

Another words JavaScript object *inherits* properties and methods from its prototype.



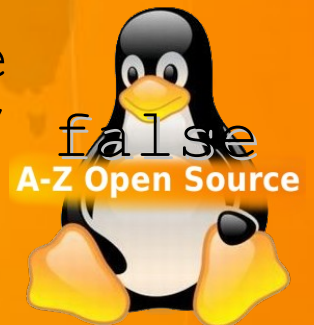
# Classes

```
function Point(iX, iY){ // Constructor function
  this.iX = iX;
  this.iY = iY;
}
```

```
Point.prototype.addToX = function(iDelta){
  this.iX += iDelta;
}
```

```
var oMyLocation = new Point(10, 20);
oMyLocation.addToX(10);
```

```
myLocation.hasOwnProperty('x'); // true
myLocation.hasOwnProperty('addToX'); // false
addToX in myLocation; // true
```



# What is with this `this`?

If the function is invoked as a method of an object `this` refers to the object the function was called on.

In regular functions `this` refers to **Global object**.

In nested functions `this` refers to **Global object**.

```
function Constr(iX) {  
  this.iX = iX;  
  var that = this;  
  helperFunction(iX+5);  
  function helperFunction(iY) {  
    that.iY = iY;  
  }  
}  
  
var oTest = new Constr(10);  
console.log(oTest.iX);  
console.log(oTest.iY);
```



# Avoid common mistakes.

- do not trash Global Object!
- always use `parseFloat()` and `parseInt(x, 10)`
- do not trash Global Object!
- always use separate statements with semicolons



# Frameworks

jQuery

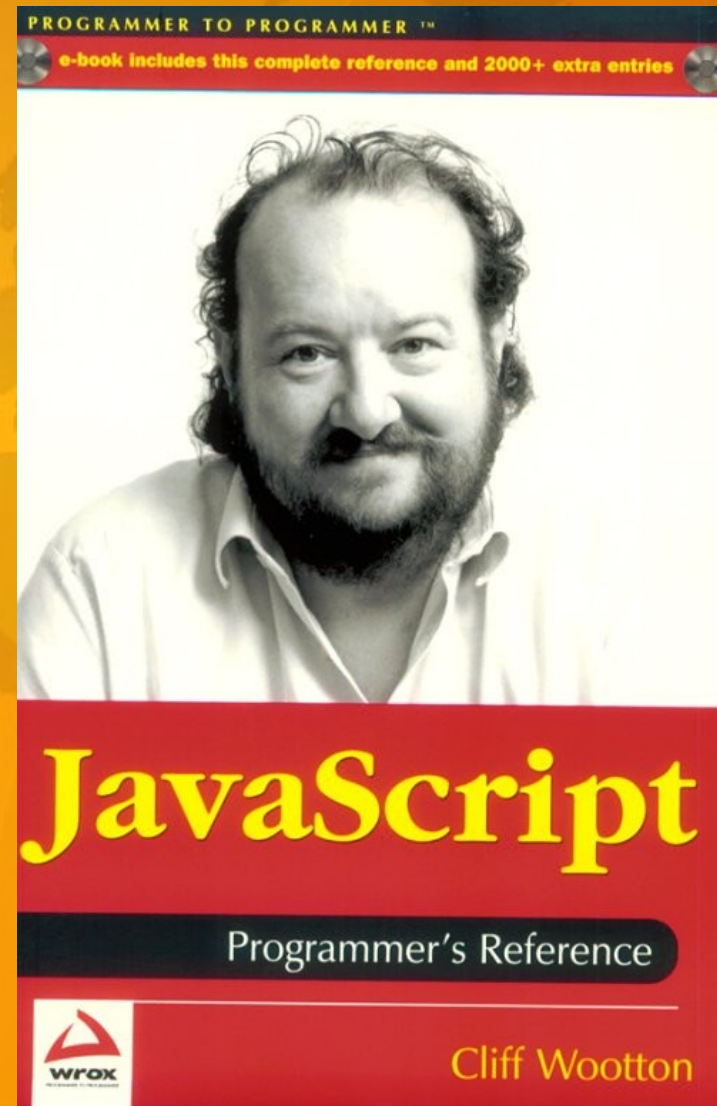
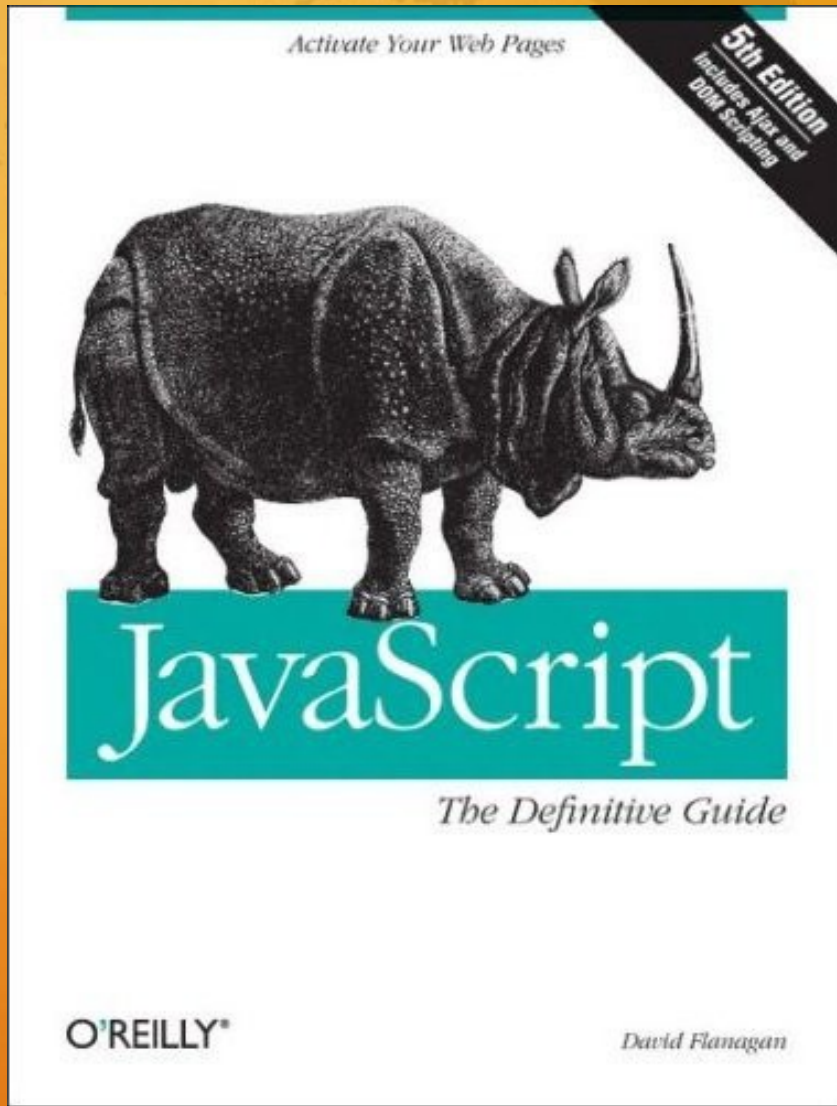
YUI

Prototype

...



# Must have books



# JavaScript

About Me:

**Ralph Zajac**

[ralph@TheOrangelT.com](mailto:ralph@TheOrangelT.com)

<http://TheOrangelT.com/presentations>

